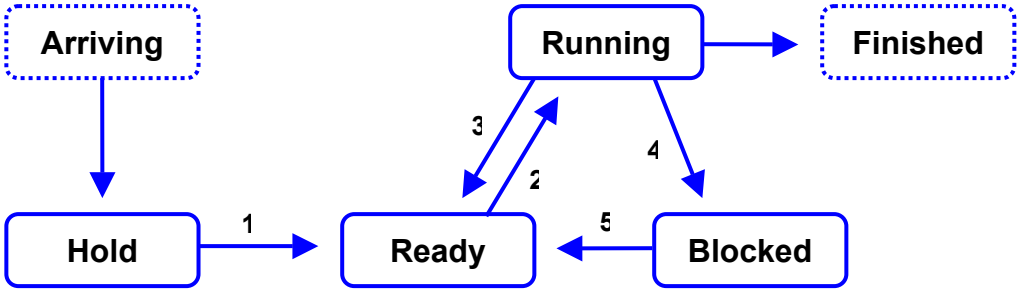
Process Scheduler Simulation

The objective of this assignment is to simulate a multiprogrammed system that uses a simple four-state scheduling strategy. The system has a long-term scheduling policy for a Hold state in which jobs are placed when they arrive initially and held until their memory requirements are fulfilled, a short-term scheduling policy for a Ready state in which jobs are placed after their memory requirements are fulfilled, a Blocked state in which jobs are placed while they wait for completion of a pending I/O operation, and a Running state for the job that is currently scheduled on the CPU. The scheduling policies for each queue are different.

Scheduler Details

The following diagram illustrates the different states a job could be in. Each of the different states and transitions is explained below.



State properties

- An arriving job that cannot get the necessary memory is put in the hold queue.
- Jobs in the Hold queue are considered strictly in FCFS order, but a job in the hold queue whose memory requirements could be satisfied with the completion of running jobs may be moved to the ready list before jobs in front of it, if those jobs requirements cannot be satisfied.
- Jobs in the ready state wait for their turn to be executed by the processor. The round robin algorithm is used to schedule these processes.
- At any given time, only one process can be in the run state.
- A process that moves to the Running state remains there for `<time quantum>` units of system time, unless it issues an I/O request or accumulates its specified CPU time.
- When a running jobs quantum expires, if there is still time remaining for that job, it is returned to the ready state.
- A process is placed in the blocked state as soon it issues an I/O request. There might be several processes in the blocked list; they are removed as their I/O requests are completed.
- A Running job that completes its execution may be "archived" in a special external state for statistical purposes.
- If two or more jobs exit one of the queues on the same system tick, they should leave in the same order they entered the state. (This only applies to certain states, obviously.)

State transition rules

In reference to the state diagram above:

1. A job makes the transition from Hold to Ready when its required memory is allocated.
2. A job makes the transition from Ready to Running if it is at the head of the ready list and a context switch occurs.
3. A job makes the transition from Running to Ready if its time quantum expires.
4. A job makes the transition from Running to Blocked if it issues an I/O request.
5. A job makes the transition from Blocked to Ready if the I/O request it is waiting for is completed.

Priority rules for event ordering

It is possible that more than one event may fall at the same time; for example, a new job may arrive at the same time that a running job terminates, freeing memory that could be used to move another job from Hold to Ready. Many other scenarios for simultaneous events are possible. If the order in which the simulation handles updating the states of entities within the

system is not fully specified then there may be multiple end results that are consistent with the system definition. We would prefer to reduce the number of acceptable variations. So…

When the system time is updated, the simulator will update the state of the system in the following order:

1. notify all relevant processes that the clock has ticked
2. add new arrivals to the hold state, if necessary (i.e., if there are new arrivals)
3. check the state of the process in the run state, if any
    a. check for process termination
    b. check for process I/O event
    c. check for quantum expiration
4. check for processes whose I/O requests have been satisfied
5. check for processes whose memory requests can now be satisfied (i.e., processes in Hold)
6. select new process to run, if necessary

Performing these steps in a different order will produce results that may be logically acceptable in a different scheduling system, but which will not satisfy the requirements of this assignment.

Assumptions and error handling

Arriving jobs that specify a memory need greater than the total primary memory of the system should be disallowed upon their arrival time. You should print an error message in this situation and discard the job.

Any syntactic or semantic errors in the input script file (described below) should be handled gracefully, including an error message and continued simulation if that is possible. Note that the point of this assignment is not to test every such scenario, and you are not expected to be particularly creative in designing such error handling.

System Operation

Your system should accept two command-line arguments, which are the names of an input file and an output file:

```
simscheduler <input file name> <output file name>
```

The input file contains all specifications concerning arrivals, I/O requests and display events.

Each line in the input file conforms to one of the specifications given below:

Comment

Any line beginning with a `'#'` character is a comment.

System initialization

The first non-comment line in the input file will always be of the form:

`sysinit` `<memory size> <time quantum>`

This line initializes the system primary memory to contain `<memory size>` number of pages each of 4 KBytes out of which 16 pages (=64 KBytes) are reserved for the kernel memory. This implicitly means that the primary memory should have greater than 64 KBytes of memory. The `<time quantum>` refers to the time-quantum used for the round robin algorithm that is used for the short-term scheduling. There will always be exactly one system initialization specification in the script file.

Job arrival

The arrival of a new job in the system is signified by a command of the form:

```
arrival <jobnum> <system time> <cpu time> <memory>
```

Here, <jobnum> is the unique identifier for this job, <system time> is the arrival time of the job, <cpu time> is the processing/service time of the job and <memory> is the number of memory pages needed by this job.

Job I/O request

A job, while it is in the running state, can perform an input/output operation. This request is in the form:

```
IOrequest <jobnum> <job time> <io time>
```

Here, <jobnum> is the unique identifier of the job doing the i/o request, <job time> is the time the request is made, and <io time> is the time for the i/o operation. Note that the job time is relative to the accumulated CPU time of the job in question.

System time advance

The system can be instructed to advance the simulation to a specified time by a command of the form:

```
simto <system time>
```

The simulation will perform all actions that should occur up to the specified system time. No additional commands will be read from the input script until this is carried out.

System state display

The user can request a snapshot of the system queues and the status of each job in the system:

```
show
```

The show command should print out the contents of each state and the statistics associated with each job, including the remaining time. The ids of the jobs completed by this time should also be printed out.

There are two (additional) variations of this command:

```
show [ blocked | hold | ready | run | terminated ]
show <jobnum>
```

Each is similar to the original but results in a display only of the specified state or of the specified job.

System shutdown

Stop processing scheduling decisions at the specified system time. Display the system queues and the status of each job in the system, handle all necessary memory deallocation and other cleanup, and terminate the simulation.

```
shutdown <system time>
```

where <system time> refers to the absolute system time at which the system shutdown is done. A full display of the final system status should be carried out before the simulation exits.

There is no guarantee that a shutdown command will be given. If not, the system should detect when all jobs have finished and the script has run out and terminate the simulation automatically in that event. In any case, after system shutdown, the arrival time, finish time, wait time, turnaround time and weighted turnaround time for each terminated job should be printed

out in tabular format. For the purposes of this assignment, use the following definitions for wait, turnaround and weighted turnaround time.

wait time	Time a process spends waiting in the ready state before its first transition to the running state
turnaround time	Time between the moment a process first enters the ready state and the moment the process exits running state for the last time (completed)
weighted turnaround time	Ratio of turnaround time to the total CPU time needed.

These two definitions imply that you do not account for the time that a job might have spent in the hold queue. Additionally, the average wait time, average turnaround time and average weighted turnaround time should be printed out.

Job numbers and all time values will be given as positive integers. There will never be two specified arriving jobs that are given the same number. Any command in the script file that specifies a system time that precedes the current simulation time when the command is read should be ignored.

The script files are guaranteed to be syntactically correct. All commands, except `shutdown`, that include a system time will be provided in ascending order. Out-of-order commands should be flagged with an error message and then ignored.

Note: All output should be sent to the output file. Assume that the tokens in the commands are separated by one or more spaces and/or tab characters.

Test Data and the Log File

Sample input and possible output data will be made available on the class site at least one week before the due date. You should create your own input files for testing your program. It is permissible to exchange test data among yourselves.

Testing may also be easier if your implementation produces extra logged output, beyond the minimal expectations stated above. You may add as much output as you like, as long as the end result is a log file that is easily scanned by someone who wants to verify its correctness.

In particular, the output produced in response to any particular command, or from reaching a specific system time, should be clearly delimited and labeled.

System Design and Implementation

You are expected to produce and implement an object-centered design for the simulator. Your implementation must be in C++, not in C. You should produce a sensible collection of classes and that each class should be assigned sensible responsibilities. It is not necessary to use inheritance, although there is no objection to doing so. Association and aggregation relationships will probably abound in a good design.

Certainly all data structures should be templates. You are free to use any STL containers you like.

You should document your implementation internally, according to the requirements given on the Programming Standards page of the course website.

Submitting Your Program

You will submit this assignment to the Curator System (read the *Student Guide*), where it will be archived for grading.

For this assignment, you must submit a gzip'd tar file containing all the source code files for your implementation (i.e., header files and cpp files). You should also submit a make file that will be used to build your program. Submit only the header and cpp files and the make file. Submit nothing else.

In order to correct submission errors and late-breaking implementation errors, you will be allowed up to five submissions for this assignment.

The Student Guide and link to the submission client can be found at: http://www.cs.vt.edu/curator/

Evaluation

The GTA will perform a build using your make file, and will run your program on a set of test data. The GTA will evaluate the correctness of your results. In addition, the GTA will evaluate your project for good internal documentation and software engineering practice.

Remember that your implementation will be tested in the McB 124 lab environment. If you use a different development platform, it is entirely your responsibility to make sure your implementation works correctly in the lab.

Note that the evaluation of your project will depend substantially on the quality of your code and documentation. See the Programming Standards page on the course website for specific requirements that should be observed in this course.

Pledge

Each of your program submissions must be pledged to conform to the Honor Code requirements for this course. Specifically, you **must** include the pledge statement provided on the Submitting Assignments page of the course website.